# 网络基础

## 网卡

又名网络适配器，定义在第二层数据链路层，作用主要是收发数据。由mac地址标识，6个字节，是物理地址。

## ip

ip用来标识一台主机  逻辑地址

iPv4 ip地址是4字节 32位

ipv6 128位   16字节

子网id   ip中被子网掩码中1连续覆盖的位

主机id   ip中被子网掩码中0连续覆盖的位

子网掩码 用来区分子网id 和主机id

192.168.1.2/24 <=> 192.168.1.2/255.255.255.0

子网id 192.168.1 主机id 2

网段地址 192.168.1.0 标识网段范围

广播地址 192.168.1.255 给网段内的ip地址发广播

网关 可以指定为192.168.1.254 同一网段的ip可以直接进行通信，不同网段的ip需要通过各网段的网关进行通信。

## 端口

作用: 用来标识应用程序(进程)

port: 2个字节 0-65535

0-1023 知名端口

自定义端口 1024 – 65535

## OSI 七层模型

### 物理层

**作用：**物理层确保原始的数据可在各种物理媒体上传输，为上层协议提供了一个传输数据的物理媒体。

**传输单元：**bit比特流

**主要功能设备/接**口**：**光纤，双绞线，中继器，集线器，网线接口

**传输协议：**IEEE 802(关于局域网和城域网的一系列标准)

### 数据链路层

**作用：**数据链路层在不可靠的物理介质上提供可靠的传输。主要作用包括物理（mac）地址寻址、数据的成帧、流量控制、数据的检错、重发。

**传输单元：**frame数据帧

**主要功能设备/接**口**：**网桥、二层交换机

**传输协议：**

* Ethernet（以太网）用于实现链路层的数据传输和MAC地址封装
* HDLC（High-level Data Link Control）高级数据链路控制，是一种面向比特的链路层协议。使用点到点链路连接，P2P 链路。
* PPP（Point-to-Point Protocol）点到点协议，直接连接在两个结点的链路上，是一种专线通信方式。

### 网络层

**作用：**网络层负责对子网间的数据包进行路由选择。此外，网络层还可以实现拥塞控制、网际互连等功能。

**传输单元：**packet数据包

**主要功能设备/接**口**：**路由器、三层交换机

**传输协议：**

* IP协议提供面向无连接不可靠传输功能，主要功能包括寻址、路由选择、分段与组装。
* ARP地址解析协议 通过ip找mac地址
* RARP反向地址解析协议 通过mac找ip

### 传输层

**作用：**传输层是第一个端到端，即进程到进程的层次。传输层负责将上层数据分段并提供端到端的、可靠的或不可靠的传输。

**传输单元：**TCP->segment报文段，UDP->Datagram数据报

**主要功能设备/接**口**：**四层交换机

**传输协议：**

* TCP
* UDP

### 会话层

**作用：**会话层管理主机之间的会话进程，即负责建立、管理、终止进程之间的会话。

**传输单元：**报文

**主要功能设备/接**口**：**--

**传输协议：**SSL安全套接字协议

### 表示层

**作用：**表示层主要解决用户信息的语法表示问题，将信息翻译成同一标准，除此以外还包括信息压缩和解压。

**传输单元：**报文

**主要功能设备/接**口**：**--

**传输协议：**--

### 应用层

**作用：**应用层为操作系统或网络应用程序提供访问网络服务的接口。

**传输单元：**报文

**主要功能设备/接**口**：**--

**传输协议：**

* HTTP超文本传输协议
* FTP文件传输协议
* SMTP简单邮件传输协议

## 网络通信过程

由主机A向主机B发送数据，则主机A的应用层将数据加上应用层协议的头部后传给传输层，传输层再加上TCP/UDP协议头部（原端口号和目标端口号）后传给网络层，网络层再加上IP协议头部（原IP和目标IP）后传给数据链路层，数据链路层再加上MAC头部（原MAC地址和目标MAC地址）后传给物理层，然后传输给主机B，再由B的数据链路层，网络层，传输层，应用层解析数据，得到主机A发送的数据。

## 网络设计模式

B/S browser/server

C/S cilent/server

c/s 性能较好 客户端容易篡改数据 开发周期较长

b/s 性能低 客户端安全 开发周期短

## 进程间通信

* **本机的进程间通信**

无名管道

命名管道

mmap

文件

信号

消息队列

共享内存

* **不同主机的进程间通信**

Socket

## MTU和MSS

* MTU(Maximum Transmit Unit)，最大传输单元，即物理接口（数据链路层）限制传输数据的大小，由硬件决定，以以太网为例，MTU=1500字节。
* MSS（Maximum Segment Size），TCP提交给IP层最大分段大小，MSS是TCP用来限制应用层最大的发送字节数。可以在TCP头部选项中发送。
* MSS值为MTU值减去IPv4 Header（20 Byte）和TCP header（20 Byte）。

# TCP和UDP高频考点

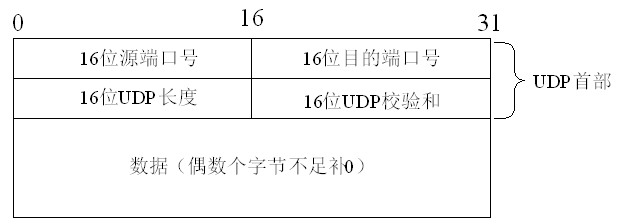
## 前言

网络层只把分组发送到目的主机，但是真正通信的并不是主机而是主机中的进程。传输层提供了进程间的逻辑通信，传输层向高层用户屏蔽了下面网络层的核心细节，使应用程序看起来像是在两个传输层实体之间有一条端到端的逻辑通信信道。

## TCP和UDP伪首部

共12字节，包括源IP（4字节），目标IP（4字节），0（1字节），协议（1字节），TCP/UDP头部长度（2字节）。

## UDP报头



报头长度：8字节

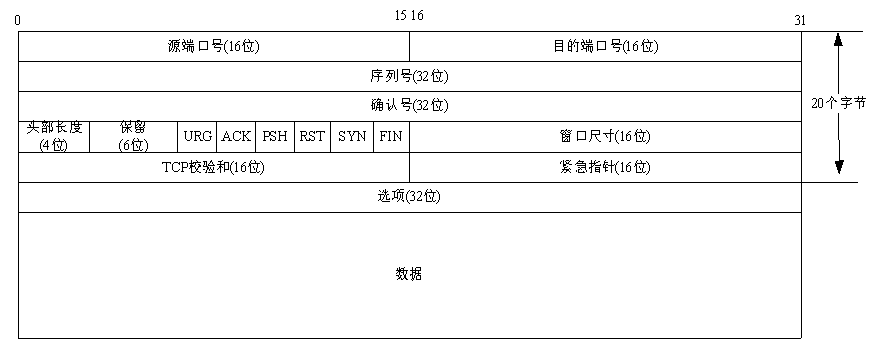
源端口号：发送方端口号

目的端口号：接收方端口号

长度：UDP用户数据报的长度，最小值是8（仅有首部）

校验和：检测UDP用户数据报在传输中是否有错，有错就丢弃。计算方式是把伪首部，头部，数据按16位划分，然后把每个划分按位相加，如果溢出则回卷（相加得到32位数，如果高16位不全为0则溢出，将高16位与低16位相加直到高16位全为0）

## TCP报头



1.源端口号：发送方端口号

2.目的端口号：接收方端口号

3.序列号：本报文段的数据的第一个字节的序号

4.确认序号：期望收到对方下一个报文段的第一个数据字节的序号

5.首部长度（数据偏移）：TCP报文段的数据起始处距离TCP报文段的起始处有多远，即首部长度。单位：32位，即以4字节为计算单位。20字节就填5

6.保留：占6位，保留为今后使用，目前应置为0

7.CWR：CWR 标志与后面的 ECE 标志都用于 IP 首部的 ECN 字段，ECE 标志为 1 时，则通知对方已将拥塞窗口缩小

ECE：若其值为 1 则会通知对方，从对方到这边的网络有阻塞。在收到数据包8.的 IP 首部中 ECN 为 1 时将 TCP 首部中的 ECE 设为 1

9.紧急URG: 此位置1，表明紧急指针字段有效，它告诉系统此报文段中有紧急数据，应尽快传送

10.确认ACK: 仅当ACK=1时确认号字段才有效，TCP规定，在连接建立后所有传达的报文段都必须把ACK置1

11.推送PSH：当两个应用进程进行交互式的通信时，有时在一端的应用进程希望在键入一个命令后立即就能够收到对方的响应。在这种情况下，TCP就可以使用推送（push）操作，这时，发送方TCP把PSH置1，并立即创建一个报文段发送出去，接收方收到PSH=1的报文段，就尽快地（即“推送”向前）交付给接收应用进程，而不再等到整个缓存都填满后再向上交付

12.复位RST: 用于复位相应的TCP连接，用来异常的关闭连接

13.同步SYN: 仅在三次握手建立TCP连接时有效。当SYN=1而ACK=0时，表明这是一个连接请求报文段，对方若同意建立连接，则应在相应的报文段中使用SYN=1和ACK=1.因此，SYN置1就表示这是一个连接请求或连接接受报文

14.终止FIN：用来释放一个连接。当FIN=1时，表明此报文段的发送方的数据已经发送完毕，并要求释放连接

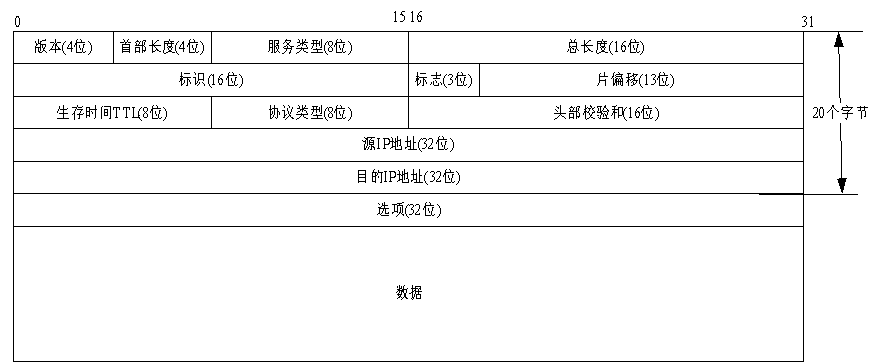
15.窗口：指发送本报文段的一方的接收窗口（而不是自己的发送窗口）

16.校验和：校验和字段检验的范围包括伪首部，首部和数据两部分

17.紧急指针：仅在URG=1时才有意义，它指出本报文段中的紧急数据的字节数（紧急数据结束后就是普通数据），即指出了紧急数据的末尾在报文中的位置，注意：即使窗口为零时也可发送紧急数据

18.选项：长度可变，最长可达40字节，当没有使用选项时，TCP首部长度是20字节

## IP报头



1.版本：IP协议的版本。通信双方使用过的IP协议的版本必须一致，目前最广泛使用的IP协议版本号为4（即IPv4)

2.首部长度：单位是32位（4字节）

3.服务类型：一般不适用，取值为0

4.总长度：指首部加上数据的总长度，单位为字节

5.标识（identification）：IP软件在存储器中维持一个计数器，每产生一个数据报，计数器就加1，并将此值赋给标识字段

6.标志（flag）：目前只有两位有意义。

7.标志字段中的最低位记为MF。MF=1即表示后面“还有分片”的数据报。MF=0表示这已是若干数据报片中的最后一个。

8.标志字段中间的一位记为DF，意思是“不能分片”，只有当DF=0时才允许分片

9.片偏移：指出较长的分组在分片后，某片在源分组中的相对位置，也就是说，相对于用户数据段的起点，该片从何处开始。片偏移以8字节为偏移单位。

10.生存时间：TTL，表明是数据报在网络中的寿命，即为“跳数限制”，由发出数据报的源点设置这个字段。路由器在转发数据之前就把TTL值减一，当TTL值减为零时，就丢弃这个数据报。

11.协议：指出此数据报携带的数据时使用何种协议，以便使目的主机的IP层知道应将数据部分上交给哪个处理过程，常用的ICMP(1),IGMP(2),TCP(6),UDP(17),IPv6（41）

12.首部校验和：只校验数据报的首部，不包括数据部分。

13.源地址：发送方IP地址

14.目的地址：接收方IP地址

## UDP特点

* UDP是无连接的，尽最大可能交付，没有拥塞控制，面向报文（对于应用程序传下来的报文不合并也不拆分，只是添加 UDP 首部），支持一对一、一对多、多对一和多对多的交互通信。
* UDP的优点：简单，轻量化。
* UDP的缺点：没有流量控制，没有应答确认机制，不能解决丢包、重发、错序问题。
* PS：并不是所有使用UDP协议的应用层都是不可靠的，应用程序可以自己实现可靠的数据传输，通过增加确认和重传机制来实现可靠性，所以使用UDP 协议最大的特点就是速度快。

## TCP特点

* TCP是面向连接的，提供可靠交付，有流量控制，拥塞控制，提供全双工通信，面向字节流（把应用层传下来的报文看成字节流，把字节流组织成大小不等的数据块），每一条 TCP 连接只能是点对点的（一对一）。
* TCP的优点：可靠，稳定，数据有序。
* TCP的缺点：效率低，占用系统资源高，容易被攻击。

## 什么是TCP三次握手？

三次握手是TCP建立连接的过程。

第一次：客户端向服务器发送SYN=1，序列号为x的报文段，客户端发送时状态变为SYN\_SEND。

第二次：服务器向客户端发送SYN=1 ACK=1，序列号为y，确认号为x+1的报文段，服务器发送时状态变为SYN\_RCVD。

第三次：客户端向客户端发送ACK=1，序列号为x+1，确认号为y+1的报文段，客户端发送时状态变为ESTABLISHED，服务器收到后状态变为ESTABLISHED。

## 什么是TCP四次挥手？

四次挥手是TCP断开连接的过程。

第一次：客户端向服务器发送FIN=1，序列号为x的报文段，客户端发送时状态变为FIN\_WAIT\_1。

第二次：服务器向客户端发送ACK=1，确认号为x+1的报文段，服务器发送时状态变为CLOSE\_WAIT，客户端收到后状态变为FIN\_WAIT\_2。

第三次：服务器向客户端发送FIN=1，序列号为y的报文段，服务器发送时状态变为LAST\_ACK。

第四次：客户端向服务端发送ACK=1，确认号为y+1的报文段，客户端发送时状态变为TIME\_WAIT，服务器收到后状态变为CLOSED，客户端等待2MSL（max segment lifetime）后状态变为CLOSED。

## 为什么握手要3次，2次和4次不行吗？

* 如果是2次握手，当客户端发送请求建立连接的报文段因为某些原因在网络结点中滞留过久，以致于过了很久才到达服务器，实际上这是一个早已失效的报文段，但服务器收到这个报文段会误以为这是客户端发出的新的建立连接的请求，便会向客户端发送确认的报文段，但客户端由于没有请求建立连接，不会理睬这个确认建立连接的报文段，此时服务器却误以为和客户端已经建立了连接，会白白浪费服务器资源。
* 如果是2次握手，客户端收到服务器确认的报文段，可以得知客户端和服务器的发送接收能力没有问题，但服务器只能得知客户端的发送能力和服务器的接收能力没有问题，但无法得知客户端的接收能力和服务器的发送能力有没有问题。但3次握手，服务器就可以得知客户端的发送接收能力，服务器的发送接收能力没问题。这样就能够建立稳定的连接，不需要第4次握手了。

## 为什么握手是3次，挥手需要4次？

因为当服务器收到客户端的SYN连接请求报文后，可以直接发送SYN+ACK报文段。其中ACK报文段是用来应答的，SYN报文段是用来同步的。但是关闭连接时，当服务器收到FIN报文段时，很可能还有数据没有发送，所以只能先回复一个ACK报文段，告诉客户端，"你发的FIN报文段我收到了"。只有等到服务器所有的数据都发送完了，才能发送FIN报文段，因此不能一起发送。故需要四步握手。

## 为什么Server端易受到SYN攻击？

SYN攻击就是客户端在短时间内伪造大量不存在的IP地址，并向服务器不断地发送SYN请求，服务器则回复确认，并等待客户端确认，由于源地址不存在，因此服务器需要不断重发直至超时，这些伪造的SYN请求将长时间占用未连接队列，导致正常的SYN请求因为队列满而被丢弃，从而引起网络拥塞甚至系统瘫痪。

解决方法：对SYN请求进行监视，如果发现某个IP发起了较多的攻击报文，直接将这个IP列入黑名单。

## 为什么TIME\_WAIT状态需要经过2MSL(max segment lifetime最大报文段生存时间)才能返回到CLOSE状态？

* 如果客户端最后一个ACK报文段丢失，服务器会重发一个FIN报文段，这时如果是CLOSE状态就无法接收报文段了，所以需要2MSL的时间来接收服务器重发的FIN报文段。
* 等待2MSL的时间可以让网络中这次连接的报文段消失，达到不影响后续连接的目的。

## 解释FIN\_WAIT, TIME\_WAIT, CLOSE\_WAIT, LAST\_ACK

* FIN\_WAIT\_1是客户端发送FIN后等待服务器确认的状态。
* CLOSE\_WAIT是服务器接收到FIN后进入的状态，该状态一般发送未发送完的数据。
* FIN\_WAIT\_2是客户端收到ACK后等待服务器发送FIN的状态，该状态半关闭，无法发送数据，但能接收数据，该状态接收服务器发送最后的数据。
* LAST\_ACK是当服务器把所有数据都发送完，应用层调用close时，向客户端发送FIN后进入的状态。
* TIME\_WAIT是当客户端收到服务器发送的FIN，并向服务器发送ACK后进入的状态，存在的意义在于防止发送的ACK丢失和清除网络中这次连接产生的数据报。

## TCP 短连接和长连接的区别

### 短连接

Client 向 Server 发送消息，Server 回应 Client，然后一次读写就完成了，这时候双方任何一个都可以发起 close 操作，不过一般都是 Client 先发起 close 操作。短连接一般只会在 Client/Server 间传递一次读写操作。

优点：管理起来比较简单，建立存在的连接都是有用的连接，不需要额外的控制手段。

应用场景：并发量大，但每个用户无需频繁操作情况下需用短连接，如WEB网站的http服务。

### 长连接

Client 与 Server 完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

在长连接的应用场景下，Client 端一般不会主动关闭它们之间的连接，Client 与 Server 之间的连接如果一直不关闭的话，随着客户端连接越来越多，Server 压力也越来越大，这时候 Server 端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可以避免一些恶意连接导致 Server 端服务受损；如果条件再允许可以以客户端为颗粒度，限制每个客户端的最大长连接数，从而避免某个客户端连累后端的服务。

优点：可以省去较多的建立连接和断开连接的时间和资源

应用场景：并发量不大，但每个用户操作频繁情况下用长连接，如数据库的连接。

## 解释RTO，RTT和超时重传

RTT（Round Trip Time）

数据从发送到接收到对方响应之间的时间间隔，即数据报在网络中一个往返用时。

RTO（Retransmission Time Out）

重传超时时间，即重传间隔，从数据发送时刻算起，超过这个时间便执行重传。

* 通常每次重传RTO是前一次重传间隔的两倍，计量单位通常是RTT。例：1RTT，2RTT，4RTT，8RTT......
* 重传次数到达上限之后停止重传

### 超时重传

超时重传保证了TCP的可靠性

发送端发送报文后若长时间未收到确认的报文则需要重发该报文。可能有以下几种情况：

* 发送的数据没能到达接收端，所以对方没有响应。
* 接收端接收到数据，但是ACK报文在返回过程中丢失。
* 接收端拒绝或丢弃数据。

## TCP粘包、拆包及解决办法

### nagle算法

* 如果包长度达到MSS，则允许发送。
* 若所有发出去的数据包均被确认，则允许发送。
* 上述条件都未满足，但发生了超时（一般设置延迟ACK，一般为200ms），则立即发送。

**作用**：是为了减少广域网的小分组数目，从而减小网络拥塞的出现。

**应用场景**：

* 对于实时性要求很高的交互上，我们不能使用nagle算法，比如FPS射击类PVP对抗类游戏。
* 对于发送接收的业务，但是每次发送的包体又是很小的，对于业务的实时性不是非常强的，我们可以使用nagle算法，将小包组成大包统一发送，减少交互次数。

### 什么是粘包、拆包？

如果通讯的一端一次性连续发送多条数据包，tcp协议会将多个数据包打包成一个tcp报文发送出去，这就是所谓的粘包。而如果通讯的一端发送的数据包超过一次tcp报文所能传输的最大值时，就会将一个数据包拆成多个最大tcp长度的tcp报文分开传输，这就叫做拆包。

### 什么情况下会发生粘包、拆包？

* 待发送数据大于 MSS（最大报文长度），TCP 在传输前将进行拆包。
* 如果使用nagle算法，要发送的数据大于 TCP 发送缓冲区剩余空间大小，将会发生拆包。
* 如果使用nagle算法，要发送的数据小于 TCP 发送缓冲区的大小，TCP 将多次写入缓冲区的数据一次发送出去，将会发生粘包。
* 接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包。

### 为什么常说TCP有粘包和拆包的问题而不说UDP？

UDP是面向报文的，UDP不会对报文作任何处理，会直接打包发送，在UDP报头中有UDP长度字段，因此每个报文的边界都是清晰的，没有粘包和拆包的问题。

### 解决方法

* 消息定长：发送端将每个数据包封装为固定长度（不够的可以通过补 0 填充），这样接收端每次接收缓冲区中读取固定长度的数据就自然而然的把每个数据包拆分开来。
* 设置消息边界：服务端从网络流中按消息边界分离出消息内容。在包尾增加回车换行符进行分割，例如 FTP 协议。
* 将消息分为消息头和消息体：消息头中包含表示消息总长度（或者消息体长度）的字段。

## TCP滑动窗口

窗口是缓存的一部分，用来暂时存放字节流。发送方和接收方各有一个窗口，接收方通过TCP报文段中的窗口字段告诉发送方自己的窗口大小，发送方根据这个值和其它信息设置自己的窗口大小。

发送窗口内的字节都允许被发送，接收窗口内的字节都允许被接收。如果发送窗口最左边的字节已经收到了确认，发送窗口就向右滑动；如果接收窗口最左边的字节已经被收到，然后发送了确认并交付主机，接收窗口就向右滑动。

如果发送方收到了一个字节的确认，那么由于有序性可知，这个字节之前的字节都已经被接收方收到。

## TCP流量控制

流量控制是为了控制发送方发送速率，保证接收方来得及接收。

接收方发送的确认报文中的窗口字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为 0，则发送方不能发送数据。

### 流量控制引发的死锁？怎么避免死锁的发生？

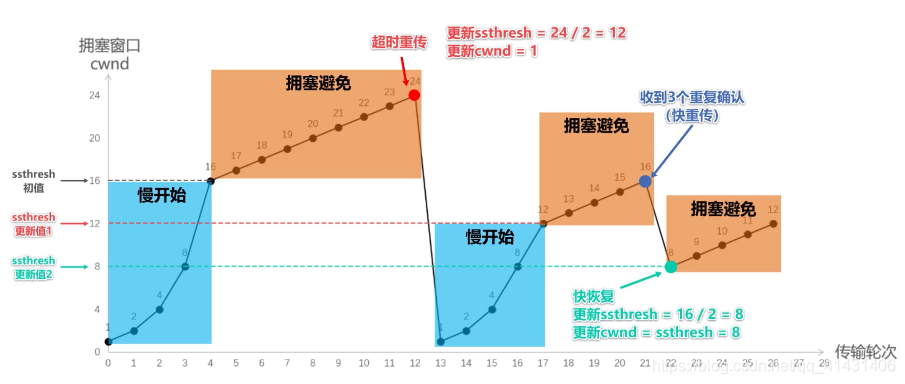
当发送者收到了一个窗口为0的应答，发送者便停止发送，等待接收者的下一个应答。但是如果接收者发送的下一个窗口不为0的应答在传输过程丢失，发送者一直等待下去，而接收者以为发送者已经收到该应答，等待接收新数据，这样双方就相互等待，从而产生死锁。

为了避免流量控制引发的死锁，TCP使用了持续计时器。每当发送者收到一个零窗口的应答后就启动该计时器。时间一到便主动发送报文询问接收者的窗口大小。若接收者仍然返回零窗口，则重置该计时器继续等待；若窗口不为0，则表示应答报文丢失了，此时重置发送窗口后开始发送，这样就避免了死锁的产生。

## 拥塞控制

如果网络出现拥塞，分组将会丢失，此时发送方会继续重传，从而导致网络拥塞程度更高。因此当出现拥塞时，应当控制发送方的速率。这一点和流量控制很像，但是出发点不同。流量控制是为了让接收方能来得及接收，而拥塞控制是为了降低整个网络的拥塞程度。

TCP 主要通过四个算法来进行拥塞控制：慢开始、拥塞避免、快重传、快恢复。发送方维持一个叫做拥塞窗口cwnd（congestion window）的状态变量。拥塞窗口与发送方窗口的区别：拥塞窗口只是一个状态变量，实际决定发送方能发送多少数据的是发送方窗口。



### 慢开始和拥塞避免

发送的最初执行慢开始，令 cwnd = 1，发送方只能发送 1 个报文段；当收到确认后，将 cwnd 加倍，因此之后发送方能够发送的报文段数量为：2、4、8 ...

注意到慢开始每个轮次都将 cwnd 加倍，这样会让 cwnd 增长速度非常快，从而使得发送方发送的速度增长速度过快，网络拥塞的可能性也就更高。设置一个慢开始阈值 ssthresh（Slow start threshold），当 cwnd >= ssthresh 时，进入拥塞避免，每个轮次只将 cwnd 加 1。

如果出现了超时，则令 ssthresh=cwnd/2，然后重新执行慢开始。

### 快重传和快恢复

在接收方，要求每次接收到报文段都应该对最后一个已收到的有序报文段进行确认。例如已经接收到M1和M2，此时收到M4，应当发送对M2的确认。

在发送方，如果收到三个重复确认，那么可以知道下一个报文段丢失，此时执行快重传，立即重传下一个报文段。例如收到三个M2，则M3丢失，立即重传M3。

在这种情况下，只是丢失个别报文段，而不是网络拥塞。因此执行快恢复，令 ssthresh=cwnd/2，cwnd=ssthresh，注意到此时直接进入拥塞避免。

慢开始和快恢复的快慢指的是cwnd的设定值，而不是cwnd的增长速率。慢开始cwnd设定为1，而快恢复 cwnd设定为ssthresh。

## 如何区分流量控制和拥塞控制？

* 流量控制属于通信双方协商；拥塞控制涉及通信链路全局。
* 流量控制需要通信双方各维护一个发送窗、一个接收窗，对任意一方，接收窗大小由自身决定，发送窗大小由接收方响应的TCP报文段中窗口值确定；拥塞控制的拥塞窗口大小变化由试探性发送一定数据量数据探查网络状况后而自适应调整。
* 实际最终发送窗口 = min{流量发送窗口，拥塞窗口}。

## TCP如何提供可靠数据传输的？

建立连接：通信前确认通信实体存在。

序号和确认号机制：确保了数据的有序性，TCP传输的字节都有序号，而在接收方的窗口中，只有按顺序的字节才会被确认，例如接收窗口中有编号为30，31，33，34的字节，那么只会发送30和31的确认，发送方会多次收到31的重复确认，这样发送方就会得知32丢失，会重发32号字节（TCP如何保证有序性）。

数据校验：校验全部数据，确保了数据的正确性。

超时重传：确保数据可以到达。

流量控制和拥塞控制：提供流量控制，避免过量发送。

## 提高网络利用率

### nagle算法

* 如果包长度达到MSS，则允许发送。
* 若所有发出去的数据包均被确认，则允许发送。
* 上述条件都未满足，但发生了超时（一般设置延迟ACK，一般为200ms），则立即发送。

**作用**：是为了减少广域网的小分组数目，从而减小网络拥塞的出现。

**应用场景**：

* 对于实时性要求很高的交互上，我们不能使用nagle算法，比如FPS射击类PVP对抗类游戏。
* 对于发送接收的业务，但是每次发送的包体又是很小的，对于业务的实时性不是非常强的，我们可以使用nagle算法，将小包组成大包统一发送，减少交互次数。

### 延迟确认应答

接收方收到数据之后可以并不立即返回确认应答，而是延迟一段时间的机制。

* 在没有收到2\*最大段长度的数据为止不做确认应答。
* 其他情况下，最大延迟 0.5秒 发送确认应答。
* TCP文件传输中，大多数是每两个数据段返回一次确认应答。

### 捎带应答

在一个TCP 包中既发送数据又发送确认应答的一种机制，由此，网络利用率会提高，计算机的负荷也会减轻，但是这种应答必须等到应用处理完数据并将作为回执的数据返回为止。

# select, poll, epoll

## 什么是多路复用？

在一个线程中同时监听多个socket网络连接。

多路: 指的是多个socket网络连接

复用: 指的是复用一个线程

多路复用主要有三种技术：select，poll，epoll。

epoll是最新的, 也是目前最好的多路复用技术。

与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。

I/O多路复用就是通过一种机制，一个进程可以监视多个描述符，一旦某个读写描述符就绪，能够通知程序进行相应的读写操作。判断同步还是异步的标准是io操作是否阻塞了当前的进程或线程。阻塞就是同步，没阻塞就是异步。但select，poll，epoll本质上都是同步I/O，也就是说这个I/O过程是阻塞的，而异步I/O则需要使用多进程，多线程等方式实现。

## 相较于多线程IO模型，IO多路复用效率更高吗？

是的，多线程的IO模型一般是主线程负责服务器的监听，每当客户端连接到来创建一个新的线程去监听是否有客户端的数据传来，这样相当于每个线程都会调用系统调用，IO多路复用是调用一个系统调用，在系统调用中通过IO事件异步唤醒的方式来监听客户端发送的数据，多线程耗费了更多的进程资源，并且使用了多次系统调用，所以IO多路复用效率更高。

## epoll是同步的还是异步的？

我理解的同步IO是指用户进程发起一个系统调用，必须等到内核IO完成后给用户进程一个返回值；异步IO是指用户进程发起一个系统调用，用户进程可以立即做别的事，内核完成IO后以信号的方式通知用户进程。

作为系统调用，epoll本身是同步的，也就是我们必须等待内核给我们返回值，但我们可以使用多线程+epoll\_wait+信号来实现异步。

## select

int select(int nfds, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

参数:

nfds：最大文件描述符+1

readfds：需要监听的读的文件描述符存放集合

writefds：需要监听的写的文件描述符存放集合 NULL

exceptfds：需要监听的异常的文件描述符存放集合 NULL

timeout：固定的时间,限时等待 NULL 永久监听

返回值：返回的是变化的文件描述符的个数

注意：变化的文件描述符会存在监听的集合中，未变化的文件描述符会从集合中删除。

**优点**: 跨平台

**缺点**:

* 文件描述符1024的限制，由于 FD\_SETSIZE的限制，如果修改需要重新编译内核。
* 只是返回变化的文件描述符的个数，具体哪个那个变化需要遍历
* 每次都需要将需要监听的文件描述集合由应用层符拷贝到内核
* 大量并发，少量活跃，select效率低

假设现在 4-1023个文件描述符需要监听，但是只有 5,1002 发来消息- 无解，只能轮询遍历。

## poll

int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);

功能: 监听多个文件描述符的属性变化

参数:

Fds：监听的数组的首元素地址

Nfds：数组有效元素的最大下标+1

Timeout：超时时间 -1是永久监听 >=0 限时等待

数组元素:

struct pollfd {

/\* file descriptor \*/ 需要监听的文件描述符

int fd;

/\* requested events \*/需要监听文件描述符什么事件 EPOLLIN 读事件 EPOLLOUT写事件

short events;

/\* returned events \*/ 返回监听到的事件 EPOLLIN 读事件 EPOLLOUT写事

short revents;

};

**poll相对与select的优缺点**

**优点:**

* 没有文件描述符1024的限制
* 请求和返回是分离的

**缺点和select一样:**

* 每次都需要将需要监听的文件描述符从应用层拷贝到内核
* 每次都需要将数组中的元素遍历一遍才知道那个变化了
* 大量并发，少量活跃，效率低

## epoll

### 创建红黑树

int epoll\_create(int size);

参数:

size：监听的文件描述符的上限，2.6版本之后写1即可

返回：返回树的句柄

### 上树 下树 修改节点

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

参数:

epfd：树的句柄

op：EPOLL\_CTL\_ADD上树 EPOLL\_CTL\_DEL下树 EPOLL\_CTL\_MOD修改

fd：上树，下树的文件描述符

event：上树的节点

typedef union epoll\_data {

void \*ptr;

int fd;

uint32\_t u32;

uint64\_t u64;

} epoll\_data\_t;

struct epoll\_event {

/\* Epoll events \*/ 需要监听的事件

uint32\_t events;

/\* User data variable \*/ 需要监听的文件描述符

epoll\_data\_t data;

};

将cfd上树

int epfd = epoll\_create(1);

struct epoll\_event ev;

ev. data.fd = cfd;

ev.events = EPOLLIN;

epoll\_ctl(epfd, EPOLL\_CTL\_ADD,cfd, &ev);

### 监听

int epoll\_wait(int epfd, struct epoll\_event \*events,

int maxevents, int timeout);

功能：监听树上文件描述符的变化

epfd：数的句柄

events：接收变化的节点的数组的首地址

maxevents：数组元素的个数

timeout：-1 永久监听 大于等于0 限时等待

返回值: 返回的是变化的文件描述符个数

## epoll优点

* 没有select最大文件描述符数量1024的限制。
* 使用共享内存（mmap）避免了将文件描述符数组从用户空间拷贝到内核空间。
* select/poll在调用时内核才会扫描监听的文件描述符。epoll采用基于事件的就绪通知方式，当监听的读写事件到来时触发回调，epoll\_wait返回。

### 为什么使用红黑树？

epoll有几个核心诉求，将要监听的文件描述符和需要监听的事件添加到内核中，这对应插入操作；将失效的文件描述符从内核中删除，这对应删除操作，当fd触发事件后，内核需要迅速找到fd并触发与其绑定的回调函数（将其放入双向链表中以便最后输出），这对应查找操作，二叉平衡树能很高效地解决上述问题，红黑树相对于二叉平衡树的插入删除做了改进。

### 为什么不使用hash表？

hash表最大的问题在于当文件描述符逐渐变多时，hash在冲突严重时需要扩容，而并发量巨大的时候扩容需要重新计算所有hash值，代价非常大。扩容可以参考redis的hash表的方案，维护2个hash表，在增删改查的过程中逐步进行扩容。

### 为什么不使用二叉堆？

二叉堆仅满足父结点大于等于或小于等于子结点的值，并没有做到排序，因此查询的时间复杂度是O(n)，不适合epoll使用。

### epoll工作方式

#### 水平触发 LT

* 读缓冲区有数据就会触发epoll\_wait
* 写缓冲区可以写就会触发epoll\_wait

#### 边沿触发 ET

* 读缓冲区数据由空转化为非空会触发epoll\_wait，也就是数据来一次只触发一次。
* 写缓冲区由满转化为不满会触发epoll\_wait。

因为设置为水平触发,只要缓存区有数据epoll\_wait就会被触发,epoll\_wait是一个系统调用，尽量少调用。所以尽量使用边沿触发，边沿出触发数据来一次只触发一次，这个时候要求一次性将数据读完（while循环读，读到最后read默认带阻塞，不能让read阻塞，因为不能再去监听，设置cfd为非阻塞，read读到最后一次返回值为-1。判断errno的值为EAGAIN,代表数据读干净）。

工作中边沿触发+非阻塞=高速模式。

# 网络API

## htonl

host to network long

将32位的本机字节顺序转换为网络字节顺序（大端）

常用于IP

## ntohl

network to host long

将32位的网络字节顺序（大端）转换为本机字节顺序

常用于IP

## htons

host to network short

将16位的本机字节顺序转换为网络字节顺序（大端）

常用于Post

## ntohs

network to host short

将16位的网络字节顺序（大端）转换为本机字节顺序

常用于Post

## inet\_pton

presentation（表达）to numeric（数值）

将点分十进制串("192.168.1.2")转成32位网络大端的数据

## inet\_ntop

numeric（数值）to presentation（表达）

将32位网络大端的数据转成点分十进制串("192.168.1.2")

## 半关闭

主动方发生在FIN\_WAIT\_2状态,这个状态时,主动方不可以在应用层发送数据了,但是应用层还可以接收数据,这个状态称为半关闭。

#include <sys/socket.h>

int shutdown(int sockfd, int how);

sockfd：需要关闭的socket的描述符

how：允许为shutdown操作选择以下几种方式:

SHUT\_RD(0)：关闭sockfd上的读功能，此选项将不允许sockfd进行读操作。

该套接字不再接收数据，任何当前在套接字接受缓冲区的数据将被无声的丢弃掉。

SHUT\_WR(1)：关闭sockfd的写功能，此选项将不允许sockfd进行写操作。进程不能在对此套接字发出写操作。

SHUT\_RDWR(2)：关闭sockfd的读写功能。相当于调用shutdown两次：首先是以SHUT\_RD，然后以SHUT\_WR。

## 心跳包

如果对方异常断开，本机检测不到，一直等待，浪费资源。

需要设置tcp的保持连接，作用就是每隔一定的时间间隔发送探测分节，如果连续发送多个探测分节对方还未回，就将次连接断开。

int keepAlive = 1;

setsockopt(listenfd,SOL\_SOCKET,SO\_KEEPALIVE,(void\*)&keepAlive, sizeof(keepAlive));

心跳包：最小粒度

乒乓包：携带比较多的数据的心跳包

## 端口复用

默认的情况下，如果一个网络应用程序的一个socket绑定了一个端口，别的socket就无法使用这个端口。

端口复用是多个应用程序使用相同的端口进行通信，绑定同一端口的socket都能发送消息，但只有最后一个绑定的socket才能接收消息。

int opt = 1;

setsockopt(fd,SOL\_SOCKET,SO\_REUSEADDR,&opt,sizeof(opt));